# 编译原理 实验 4

## 目标代码生成(OC)

计算机科学与技术系

黄彬寓，161220047，161220047@smail.nju.edu.cn

实验情况：

完成了实验讲义中要求的生成目标代码，并写入.s 文件中。

额外完成要求：

无。

数据结构：

```
typedef struct DataAddr_ DataAddr;
typedef struct Reg_ Reg;
typedef struct DataAddrList_ DataAddrList;

struct DataAddr_
{
    enum { DATA_TEMP,DATA_VAR }kind;
    int no;
    int offset;
};
struct Reg_
{
    bool unused;
    char name[8];
    DataAddr* da;
};
struct DataAddrList_
{
    DataAddr *da;
    DataAddrList* next;
};

extern Reg regs_t[10];
```

DataAddr 结构体代表栈中的某一地址，其存储的对应于中间代码中的变量/临时变量的类型和序号(如 t3，v6)和该变量对应于 ebp($fp)的地址偏移量。

Reg 结构体代表$t0-$t9 这十个寄存器，其可能被使用可能未被使用，当被使用

时代表其在内存中有值相对应的地址。

DataAddrList 结构体代表当前函数体中已经存入内存中的变量/临时变量的集合，用链表表示。

## 实现方法：

采用朴素分配算法，即将所有的变量和临时变量都存放到内存中，每次运算前把操作数从内存取到寄存器中，运算完成后把结果从寄存器保存到内存中。

基于实验三生成的中间代码，顺次读取每条中间代码并生成相应的目标代码。

我将中间代码大致分为两类：

1.与函数调用无关的代码，2.与函数调用有关的代码。

与函数调用无关的代码中，大致就是取数据于寄存器，修改值，保存回数据。

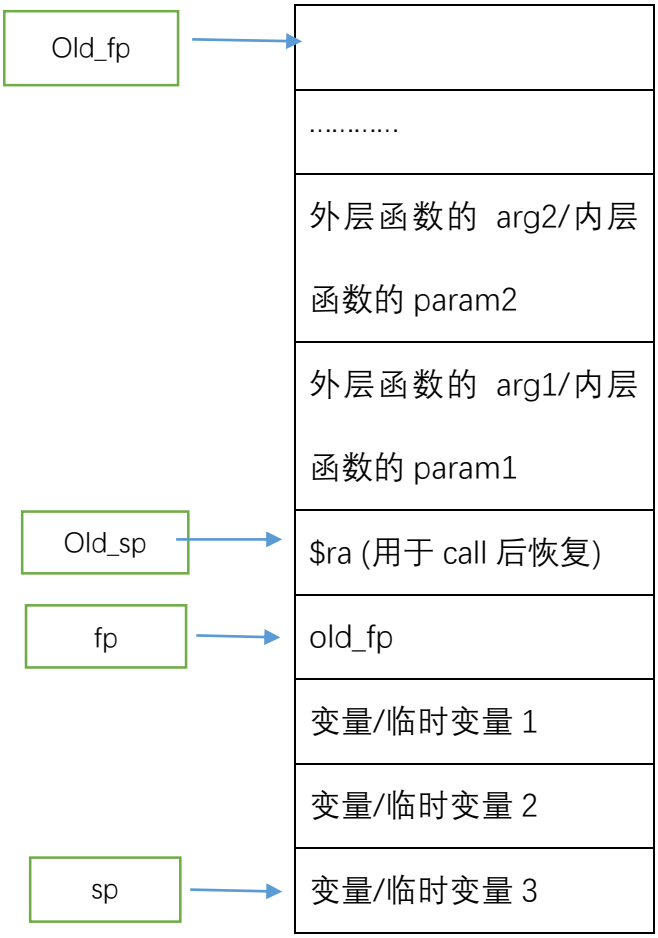
如加减乘除：

```
case IC_PLUS: case IC_MINUS: case IC_STAR: case IC_DIV:
{
    Reg* rr = get_reg(&code->u.binop.result);
    Reg* r1 = get_reg(&code->u.binop.op1);
    Reg* r2 = get_reg(&code->u.binop.op2);
    if(code->kind == IC_PLUS)
        printf("  add %s, %s, %s\n",rr->name,r1->name,r2->name);
    else if(code->kind == IC_MINUS)
        printf("  sub %s, %s, %s\n",rr->name,r1->name,r2->name);
    else if(code->kind == IC_STAR)
        printf("  mul %s, %s, %s\n",rr->name,r1->name,r2->name);
    else
    {
        printf("  div %s, %s\n",r1->name,r2->name);
        printf("  mflo %s\n",rr->name);
    }
    spill_reg(rr);
    free_reg(r1);
    free_reg(r2);
    break;
}
```

1. 读取操作数到寄存器中。其中使用了 get_reg 函数，在该函数中会判断该变量/临时变量是否在栈中，若在则直接取出，若不在，则将 esp 减 4 把该操作数的数据放入栈中，然后从栈中取出。
2. 生成目标代码指令。

3. 释放寄存器。如果是其值在寄存器中遭到修改，如赋值语句的左值，则调用 spill_reg 表示先将寄存器中的值回写进栈再释放寄存器，否则，调用 free_reg 直接释放寄存器。

与函数调用有关的代码是这次实验的难点，但只要理清自己需要在调用函数前后保存或恢复哪些数据，已经知道加上这些需要保存或恢复的数据后栈的结构是怎么样的，就比较好处理。




在我的栈中，我讲每个函数的栈划分成这样，中间两个地址$fp 及 $fp+4 分别用于函数结束后的恢复。更大的地址$fp+8 及以上是该函数传入的实参地址 param。更小的地址$fp-4 及以下是该函数体内部的变量/临时变量。其中，实参+变量/临时变量共同构成了该函数体的 DataAddrList 中的链表信息。

## 编译运行方法：

环境：gcc 6.3.0 Flex 2.6.1 Bison 3.0.4

(1)在 Code 目录下输入指令 make

(2)将生成的 parser 程序拖至根目录下

(3)在根目录下输入指令 ./parser Test/lab4_cmm/testX.cmm Test/lab4_s/testX.s

(4)打开 Test/lab4_s 文件，生成的 asm 文件就在该目录下。可以使用 QtSPIM 程序进行测试。

## 实验总结：

由于这次实验是考完期末考试之后写的，心态上有所松懈，所以实验写的比较赶。总的来说较为简陋，没有做足够的优化，仅能确保答案的正确性。